Secure, Dynamic and Distributed Access Control Stack for Database Applications

Óscar Mortágua Pereira[1], Diogo Domingues Regateiro[2], Rui L. Aguiar[3]
Instituto de Telecomunicações
DETI, University of Aveiro
Aveiro, Portugal
{omp[1], diogoregateiro[2], ruilaa[3]}@ua.pt

Abstract— **In database applications, access control security layers are mostly developed from tools provided by vendors of database management systems and deployed in the same servers containing the data to be protected. This solution conveys several drawbacks. Among them we emphasize: 1) if policies are complex, their enforcement can lead to performance decay of database servers; 2) when modifications in the established policies implies modifications in the business logic (usually deployed at the client-side), there is no other possibility than modify the business logic in advance and, finally, 3) malicious users can issue CRUD expressions systematically against the DBMS expecting to identify any security gap. In order to overcome these drawbacks, in this paper we propose an access control stack characterized by: most of the mechanisms are deployed at the client-side; whenever security policies evolve, the security mechanisms are automatically updated at runtime and, finally, client-side applications do not handle CRUD expressions directly. We also present an implementation of the proposed stack to prove its feasibility. This paper presents a new approach to enforce access control in database applications, this way expecting to contribute positively to the state of the art in the field.**

Keywords-information security; access control; database; SQL; software architecture.

I. INTRODUCTION

Access control [1], [2] is a critical security issue in many software systems. Access control *"is concerned with limiting the activity of legitimate* users." [3]. Basically, it is a process to supervise every user's requests to access protected resources, in our case data residing inside database management systems (DBMS), by determining whether authorizations should be granted or denied. Access to data stored in DBMS is mostly achieved issuing Create, Read, Update and Delete (CRUD) expressions from client-side applications. Nevertheless, access control security layers are traditionally deployed in centralized servers where the data to be protected are stored. Rephrasing, the attackers are distributed and the security wall is built around the data. Very few scientific contributions have been proposed based on a distributed architecture. Basically, in a distributed architecture the walls are not centralized but deployed by putting them close to the attackers, as in [4], [5]. At first sight, the distributed architecture would seem the best one. Although, distributed architectures raise additional security concerns, such as how to trust that legitimate users are issuing authorized requests only. To overcome this security concern, in this paper we propose a security stack, herein referred to as Secure, Dynamic and Distributed Access Control Stack for Database Applications. The security stack comprises the policies to be enforced and also the correspondent authorized Create, Read, Update and Delete (CRUD) expressions. In order to keep access control mechanisms always aligned with the established policies, they are automatically updated at runtime whenever policies are modified. We also provide an empirical proof of concept to demonstrate the feasibility of the proposed security stack. It is expected that the outcome of this paper can contribute to new approaches to enforce access control policies, namely by deploying them based on distributed architectures.

The remainder of this paper is organized as hereafter described. Chapter II presents the motivation for the work detailed in this paper, chapter III presents the related work, chapter IV presents the security stack, chapter V presents a proof a concept of the security stack and chapter VI concludes this paper and details the future work.

II. MOTIVATION

Currently, there is no standard to enforce access control policies in database applications. Security experts complement security layers built from embedded tools provided by vendors of DBMS with additional security artifacts (components and/or hard-coded inside the client-application) built from scratch and tailored to the specific scenario. Approaches based on this approach convey several drawbacks. Among them we emphasize:

Scalability: security layers based on DBMS tools share concurrently the computational resources allocated to DBMS. When access control policies are many and complex, they can lead to performance decay. Moreover, very often the security layers resort to additional techniques, such query rewriting techniques [6][7][8][9][10], the use of views [10][11][12][11][12][6] and parameterized views [15]. Inevitably, these additional artifacts will lead to the allocation of additional computational resources and, therefore, additional performance decay, as explicitly recognized in most of the papers by their authors.

Maintainability: security layers built from DBMS tools can be easily maintained since they are centralized in a server. In

(DOI reference number: 10.18293/SEKE2015-049)

opposite, security artifacts deployed in client-side applications can lead to huge maintenance efforts. For example, if an attribute of a table becomes protected by a new or modified policy, there is no other alternative than modify in advance all the CRUD expressions violating that policy. Otherwise, CRUD expressions are rejected while the updating process is not completed.

Security gap: CRUD expressions are mostly issued from client-side applications. If this process is not controlled, malicious users can issue CRUD expressions systematically against the DBMS expecting to identify any security gap. It is very likely that professional malicious users will end up violating the protected data to some extension. Malicious users can follow additional malicious approaches, such as SQL Injection, the use of sequences of valid CRUD expressions [16] and also resorting to reflection [17] to get and/or modify the way software works. Other security gaps can arise from collecting data sent between a legitimate user and the DBMS and also from personification.

To overcome these drawbacks, we propose a security stack with the following properties: 1) access control mechanisms are mostly distributed in each client-side system; 2) access control mechanisms are updated whenever modifications occur in the established policies; 3) client-side applications are prevented from issuing CRUD expressions and, finally, 4) additional security mechanisms are provided for user authentication and secure connections.

III. Related Work

We will now discuss the work in the field of access control enforcement and how they relate to our proposed security stack.

A complete architecture for web applications is presented in [18], where the problem of sensitive data being stored in a browser is solved by enforcing end-to-end security on data, across the virtual machine, operating system, networking and application layers. However, it relies solely on mandatory access control to enforce the end-to-end security policies and it is only concerned with web applications.

In [19] a new tool is presented, Ur/Web, where the access control policies can be checked by CRUD expressions written by programmers in a RDBMS backed system. It uses an extension to the standard SQL language with predicates that indicates 'which secrets the users knows' and determine what information can be disclosed. However, these predicates are not checked against the access control policies, potentially leaking protected information. λ_{DB} [20] is a programming language that enforces access control policies to data by static typing for data-centric programs. It allows the definition of entities that are checked at compile-time with the defined access control policies. Each entity has a set of attributes that are given a read and write permissions with different predicates, similarly to Ur/Web. Another similar work is presented in [21] using predicated grants. These solutions only provide access control mechanisms, not addressing the rest of the stack.

The work presented in [22] aims to provide role-based access control using proxy objects, generated using a custom compilation tool. Each role has a set of different proxy objects, which are made available through Java Remote Method Invocation (RMI). Note that the proxy objects only implement the methods that the given role can execute, therefore it is not possible for a client to execute other methods. This method guards against reflection mechanisms since they do not work over sockets, but this solution is limited to RBAC and it protects access to java objects, not the data layer itself. Similarly, [23] is a security-typed programming language that extends Java that aims to give support for information flow and access control, enforced at both compile and runtime. However, this solution is also mostly used to manage information at the application level, leaving out other data sources, like a RDBMS.

In [4], the authors present a proposal to extend the RBAC model to control sequences of CRUD expressions. In [5], the authors present a proposal to implement distributed RBAC mechanisms. The content of both is relevant but they are focused on RBAC policies only. Additionally, key issues such avoiding the use of CRUD expressions at the client-side is also not supported.

IV. Access Control Stack Presentation

For a software solution to provide secure, dynamic and distributed access control mechanisms we need to evaluate the requirements and the problems that arise from this architecture.

A. Access Control Stack

We will now discuss the necessary requirements to build Secure, Dynamic and Distributed Access Control Stack for Database Applications. As previously described, the identified fragilities of current solutions, and therefore to be addressed by the stack, are: scalability, maintainability and security gap. Figure 1 presents the general access control stack.

The data layer will obviously reside on the server side, so that it can be provisioned to all the clients. It can use relational DBMS[24] or some other form of data storage, e.g. a distributed file system[25][26] (e.g. Apache Hadoop[27]).

The application layer requesting access to the protected data resides on the client-side. It will access the data layer through the Security Layer.

The Security Layer is the layer responsible to ensure that all operations requested by the Application layer follow the established security rules. It comprises three main components: Security Manager, Access Control and Network Security.

Security Manager: The Security Manager component needs to address three main issues. The first one is to ensure that access control mechanisms are dynamically built and updated at runtime. This is very important to overcome one of the fragilities of current solutions: maintainability. The building and updating processes need to be based on an automated engine responsible for generating the necessary code for the Access Control component. The automated engine takes as input the policies to be enforced and also the architectural model to be implemented, as shown in Figure 2. The place to store the policies to be enforced depends on each particular use case or scenario. The second one is ensure that the access to the Data layer follows the established security rules. The third and last one is provide a standard interface to Application layer.

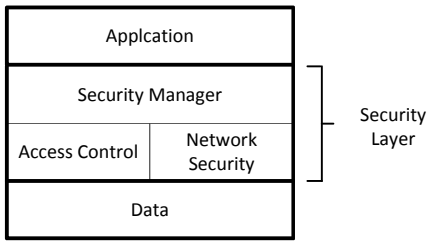

Figure 1. General access control stack.

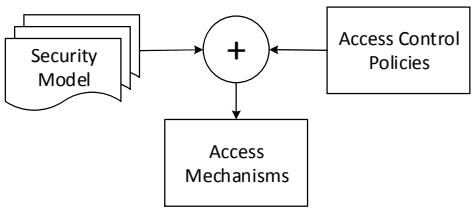

Figure 2. Access control mechanisms generation process.

This is quite relevant because Access Control component is not static as previously described. It depends on the policies to be enforced.

Network Security: Connections to database need to follow some security rules, namely authentication and secure connections. These rules are enforced by the Network Security component. Authentication mechanism forces users to present some sort of identification before accessing the data. To prevent malicious users from accessing the data when it is sent to the authenticated users it should be possible to setup a secure communication channel, which is typically done using the SSL/TLS [28] communication protocols.

Access Control: The architecture and functionalities of the Access Control component depend on the policy to be enforced and also on the architectural model to be implemented. Nevertheless, independently from the policy to be adopted, the drawbacks related to security gaps need to be addressed. Three main functionalities are required, as shown in Figure 3. The first one is the type of policy to be used, such as RBAC. The second one is the use of Sequence Controllers which are responsible to only allow the execution of valid sequences of CRUD expressions. The third one is the deployment of CRUD pointers instead of CRUD expressions. The first functionality enforces the top level functionalities of the policy to be enforced. The second functionality prevents malicious users from issuing sequences of authorized CRUD expressions to disclose the protected data. The third functionality prevents malicious users from resorting to techniques based on CRUD expressions to violate the established policies.

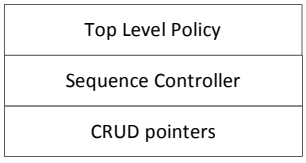

Figure 3. Access Control sub-components.

B. *Technical Issues*

Like any other software solution, there are technical issues that need to be solved before it can be expected to be considered useful. As such, we will now discuss the issues related to the Security Layer. However, it is impossible to discuss every issue that can rise from every possible scenario, so this chapter should be used as a starting point and the scenario-specific issues analyzed separately.

The Security Layer handles the Security Manager, the Access Control and the Network Security components. The Network Security aims at addressing two security problems: the network communication and the client authentication. The access control layer is where the access control effectively happens. It must be dynamic and distributed, which means that it must adapt to policy changes and be enforced on the client side. The Security Manager configures and manages the Network Security and the access control mechanisms.

Regarding the access control, the first problem comes from the fact that it is distributed. This means that the access control mechanisms will be subject to all kinds of attacks by malicious users that cannot be detected, since it happens on the client side. Therefore, there must be some mechanisms in place in order to prevent them. One of the major concerns are the reflection mechanisms[17] that some programming languages provide. These allow a program to inspect and modify the structures and behavior of the program at runtime (specifically the values, meta-data, properties and functions). This means that a software solution based on this stack cannot blindly rely on the distributed access control mechanisms to stop malicious users. To address this problem the access mechanisms were provided with CRUD pointers. CRUD pointers are some identifying tokens that are used by the client application instead of the actual CRUD expressions, which were pushed to the server-side. By making these pointers hard to guess and valid for a finite period of time, which must be small enough to prevent other users from using it by guessing, the usage of reflection mechanisms no longer threatens to manipulate the CRUD pointers.

The dynamic counterpart of the access control layer requires that the access control mechanisms in the client applications change as the access control policies change. This requires the clients to be notified somehow when they change and to enforce the changes immediately. To achieve this we have a Security Manager that implements the access mechanisms using a security model that defines how the access control mechanisms should be created from the access control policies. It also provides them with the CRUD pointers received from the server and handles the network security procedures on behalf of the client application.

Finally, the network security has many problems to address, of which we will emphasize two: the problem that the data sent between two entities in a network can be read by anyone if no measures are taken to prevent it, and the problem that the entities involved generally do not prove their identity, allowing impersonation to occur. The first problem does not make it possible to the malicious user to manipulate the data being sent. However, sensible data (e.g. a client's identification) can still be acquired, which poses a serious security breach. The second

problem not only allows the data that is sent to be manipulated but also has all the problems the first problem implies, making it a greater concern. A common approach to handle this is the usage of SSL/TLS protocols, which can create a secure communication channel over an insecure medium, like the internet. SSL/TLS protocols verify the identity of servers and optionally also of users. Authentication of users is usually handled by a server side application that receives the client's identification tokens (e.g. a username and password) and provides the client application with the means to access the data if the tokens are valid.

V. PROOF OF CONCEPT

With this proof of concept we intend to demonstrate that the proposed access control stack is feasible of getting implemented. To achieve this we used a Java application as the client and the SQL Server 2010 RDBMS to manage the sample data we used. The implementation is called S-DRACA, which stands for Secure, Dynamic and Distributed Role-based Access Control Architecture.

A. Overview

We will now give an overview of the several components that are part of S-DRACA.

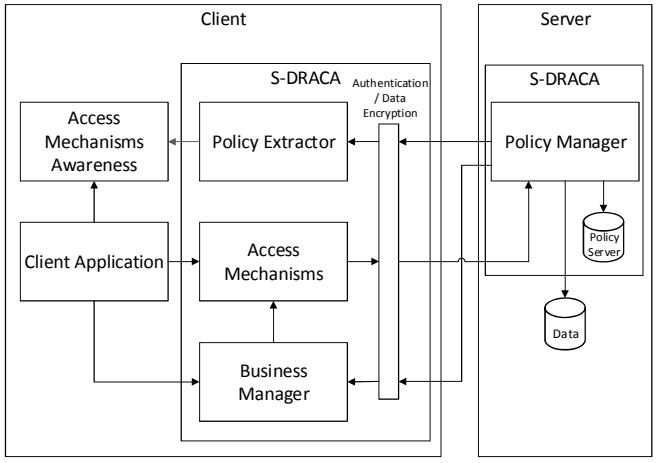

Figure 4. S-DRACA block diagram.

Figure 4 shows the block diagram of our proof of concept. We can see that on the server side we store the data in a database, along with the information about the access control policies in place on the system, which was stored in the Policy Server database. There is also a server application, the Policy Manager, which implements the SSL/TLS encryption and authentication mechanisms and also manages the policies to enforce in the system. In our proof of concept we only had one instance of a Policy Manager, but there could be many over different servers to distribute load in a cluster. On the client side we have the access mechanisms, which the client application uses to access the data stored on the server. The access mechanisms are instantiated by a Business Manager, which follows the policies defined on the server. We also have a Policy Extractor, a custom java annotation, which creates interfaces that the client application can use to access the data, using the Security Model and the defined access control policies.

The access control policy used in our proof of concept is the role-based access control (RBAC). We chose this type of access control because it is natively supported by many DBMS, but the model used in the Policy Server could be changed to implement any type of access control policy without any implication to our proof of concept.

We will now explain how each layer was implemented in S-DRACA.

B. Layer Implementation

In S-DRACA, each layer was implemented independently for each other. The data layer is managed by the SQL Server 2010 RDBMS and it is accessed by the Policy Manager. It stores the data being protected as well as the access control policies defined for the system. Any operation requested by the client-application to manipulate the defined policies must go through the Policy Manager, to which the client application must be connected and authenticated. The requests to access the data are sent directly to the RDBMS, reusing the secure communication channel created when the client application connected to the Policy Manager to authenticate.

The network security component is implemented both on the client (Authentication and Data Encryption block), and on the server side (Policy Manager block). It uses several standards of the industry for data encryption and authentication, such as SSL/TLS and using hashed and salted passwords to store the client's credentials, respectively.

For the remainder of the security layer, we had to resolve the problems that originated from the programming language's reflection features and make it adapt dynamically to changes made to the policies defined in the system. Our access mechanisms are Java classes, called Business Schemas, see Figure 5, which only implement functions to access and manipulate data that the client application is allowed to. This is possible because when the client application authenticates with the Policy Manager, it receives the policies stored in the Policy Server that applies to said client. The Business Manager then

```
100     S_Cust = ss.businessService(Role_IRole_B1.s_customers,
101         Role_IRole_B1.s_customers_S_Customers_byCountry);
102     S_Cust.execute(country);
103     if(S_Cust.moveNext()) {
104         custName= S_Cust.CustomerName();
105         // read more attributes
106         // ... code
107         S_Cust.beginUpdate();
108         S_Cust.uCustomerName(custName);
109         // update more attributes
110         S_Cust.updateRow();
111         // ... code
112         S_Cust.beginInsert();
113         S_Cust.iCustomerName(custName);
114         // insert more attributes
115         S_Cust.endInsert(true);
116         // ... code
117         S_Cust.deleteRow();
118     }
```

Figure 5. S-DRACA usage example.

uses this information and the Security Model to generate the Business Schemas with the appropriate methods (see Figure 2). The client application can then use these runtime generated Business Schemas because they implement interfaces generated by the Policy Extractor during compilation, known as Access Mechanisms Awareness, which also follow the Security Model. This prevents the clients from requesting operations that they do not have access to, but reflection mechanisms can still expose the private connection objects the Business Schemas use to request the operations. We have solved this issue using the CRUD pointers approach. Figure 5 shows a simple example of the core interface S-DRACA provides to the developers. The Business Schema S_Customers is instantiated (line 100) and executed (line 102) to obtain the data from the database. Then, if some data is returned, information can be read (line 104), updated (lines 107-110), inserted (lines 112-115) and deleted (line 117).

Note that not every user might see all the operations as shown in Figure 5. Since the Business Schemas are created dynamically from the access control policies retrieved from the server, only the authorized operations are actually implemented. This prevents developers from performing an operation they are not allowed to, which would only be known at runtime and could even go unnoticed for a long time if those operations are issued under rare circumstances.

To guarantee that the access control mechanisms adapt to changes made to the defined access control policies, the database notifies the Policy Manager, through the use of triggers, when and what information in the Policy Server was inserted, deleted or altered. This prompts the Policy Manager to verify which clients must be notified of the changes and send them the modifications. The Policy Manager of each client, upon receiving the changes, re-implements the Business Schemas and loads them, which takes immediate effect on current and future instantiations. However, the interfaces created by the Policy Server cannot be modified, since they were created during compilation, so until the client application is updated it will generate errors when methods that are no longer accessible are invoked.

Finally, we discuss the optional CRUD sequencer component, see Figure 6. We allowed the definition of sequences of Business Schemas on the policy model that is used in the policy server, which can be turned on or off at any time. This meant that the client application has to follow a particular set of generated Business Schemas if it wants to perform some operation. To ease the development of applications, the Policy Extractor also creates a java class that uses the Business Manager to create instantiations of the first Business Schema of each sequence (*factory* object at lines 74 and 80). Then, each Business Schema has a method to instantiate the next one in the sequence (lines 76 and 81). We also allow the definition of sets of rules when a client moves in a sequence in the access control policy, e.g. moving from the first Business Schema in a given sequence to the second could prevent the client application from using the first one again.

This implementation of the CRUD sequencer has a couple of problems, however. First, to make sure that it can adapt to a large number of scenarios the sequence definition model must

```
74 S_Cust_1 = factory.get_S_Customers_all(session);
75 S_Cust_1.execute();
76 S_Orders = S_Cust_1.nextBE_S1(
77     Role_IRole_B1.s_orders_S_Orders_byShipCountry,
78     session);
79 S_Orders.execute(S_Cust_1, "Portugal");
80 S_Cust_2 = factory.get_S_Customers_all(session);
81 I_Orders = S_Cust_2.nextBE_S2(
82     Role_IRole_B1.i_orders_I_Orders_withCostumerID,
83     session);
```

Figure 6. S-DRACA CRUD sequence usage.

be flexible. Secondly, when the same Business Schema is used in more than one sequence it can potentially have a "next" method for each sequence it belongs to. Work is being done to address these problems.

Figure 6 shows the interface made available to the developers to use the CRUD sequencer. The factory class (lines 74 and 80) allows to obtain the first Business Schema in a sequence. When all the operations on that Business Schema are performed, the next Business Schemas can be requested using the "next" method (lines 76 and 81). These Business Schemas can be used normally as shown in Figure 5.

C. *Performance Assessment*

In order to evaluate the overhead induced by our access control stack solution, a performance assessment was carried out. Basically, we compared the initialization, instantiation and response time between the traditional solution using JDBC and the solution proposed in this paper (pushing the CRUD expressions to the server). We measured the time it took for the system to be ready to be used (obtain the connection object in JDBC or a Business Schema in S-DRACA), to execute a single Select expression when the server has 500 CRUD expressions stored, and for changes on the access control policy to be applied on the clients, along with the bandwidth used. We did not time the network security and authentication features, since they should always be implemented if the use case requires it, whether the proposed stack is used or not. For the adaptation process, we modified the access control policies 1000 times and measured the time it took for the changes to take effect on the clients and the bandwidth used.

The two solutions were implemented and tested in a PC with Windows 7 Enterprise and no network connection to prevent delays. The data was stored using SQL Server 2010 and all unnecessary processes were shut down. We verified the time it took for both solutions 10.000 times. Additionally, the select statement was executed on a table with 100 and on another with 100.000 records. We also cleared the DBMS cache between each execution.

For the initialization, the JDBC solution only took about 12 ms to provide a connection object, while the S-DRACA solution took 2905 ms. This time is explained with the initial configuration that is needed: requesting the access control policies, generate the access control mechanisms and instantiate them. Although this process takes a significant amount of time, it is only required once per session at the start. Regarding the execution of the select statement, the table with 100 rows showed an average increase of 4 ms when using the

solution proposed in this paper, from 1ms to 5ms. When targeting the table with 100.000 rows, the select statement took, on average, 1745ms to execute the CRUD statement directly while the solution proposed in this paper took 1750ms, showing a 5ms increase. The dynamic adaptation process took 10 ms on average to complete and around 350 bytes of bandwidth per Business Schema authorized, which also has the associated CRUDs information, and only 50 bytes if revoked. We can conclude, then, that the overhead introduced with our proposal is very small and can in most cases be neglected.

VI. Conclusion and Future Work

In this paper we presented an access control stack that aims at providing distributed and dynamic access control mechanisms that enforce the access control policies on the client side while maintaining the system secure. The main drawbacks of current solutions were identified: scalability, maintainability costs and security gaps. Scalability issues were overcome by deploying most of the access control mechanisms in client-side systems. Maintainability issues were overcome by providing automated processes to dynamically update the distributed access control mechanisms at runtime. The security gaps were overcome by: 1) implementing Sequence Controllers, 2) preventing client applications from directly using CRUD expressions and, finally, 3) by using secure connections between clients and DBMS. We also provided a proof of concept to empirically demonstrate that the presented solution is feasible.

A performance assessment has also been carried out to evaluate the impact of our proposal. The collected results show that its impact is unnoticeable when executing CRUD expressions or changing the access control policies. In opposite, the establishment of secure connections induced a significant overhead. Nevertheless, we cannot forget that this process is executed only once in each session. As a final conclusion, the stack herein presented shows to be a promising approach to overcome the identified drawbacks of most of the current approaches to enforce access control policies.

VII. References

[1] P. Samarati and S. D. C. di Vimercati, "Access Control: Policies, Models, and Mechanisms," in *Foundations of Security Analysis and Design (LNCS)*, vol. 2171, Springer, 2001, pp. 137–196.

[2] S. D. C. di Vimercati, S. Foresti, and P. Samarati, "Recent Advances in Access Control - Handbook of Database Security," in *Handbook of Database Security*, M. Gertz and S. Jajodia, Eds. Springer, 2008, pp. 1–26.

[3] R. S. Sandhu and P. Samarati, "Access Control: Principle and Practice," *Commun. Mag. IEEE*, vol. 32, no. 9, pp. 40–48, 1994.

[4] Ó. M. Pereira, D. D. Regateiro, and R. L. Aguiar, "Extending RBAC Model to Control Sequences of CRUD Expressions," in *26th Intl. Conf. on Software Engineering and Knowledge Engineering*, 2014.

[5] Ó. M. Pereira, D. D. Regateiro, and R. L. Aguiar, "Role-Based Access Control Mechanisms Distributed, Statically Implemented and Driven by CRUD Expressions," in *ISCC'14 - 9th. IEEE Symposium on Computers and Communications*, 2014.

[6] Oracle, "Using Oracle Virtual Private Database to Control Data Access," 2011. [Online]. Available: http://docs.oracle.com/cd/B28359_01/network.111/b28531/vpd.htm#CIHBAJGI.

[7] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt, "Limiting disclosure in hippocratic databases," *30th Int. Conf. on Very Large Databases*. VLDB Endowment, Toronto, Canada, pp. 108–119, 2004.

[8] Q. Wang, T. Yu, N. Li, J. Lobo, E. Bertino, K. Irwin, and J.-W. Byun, "On the correctness criteria of fine-grained access control in relational databases," *33rd Int. Conf. on Very Large Data Bases*. VLDB Endowment, Vienna, Austria, pp. 555–566, 2007.

[9] S. Barker, "Dynamic Meta-level Access Control in SQL," *22nd Annual IFIP WG 11.3 Working Conf on Data and Applications Security*. Springer-Verlag, London, UK, pp. 1–16, 2008.

[10] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending Query Rewriting Techniques for Fine-grained Access Control," *ACM SIGMOD Int. Conf. on Management of Data*. ACM, Paris, France, pp. 551–562, 2004.

[11] J. Eder, "View Definitions with Parameters," *2nd Intl Workshop on Advances in Databases and Information Systems*. Springer-Verlag, pp. 170–184, 1996.

[12] Y.-J. Hu and J.-J. Yang, "A semantic privacy-preserving model for data sharing and integration," *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*. ACM, Sogndal, Norway, pp. 1–12, 2011.

[13] L. E. Olson, C. A. Gunter, and P. Madhusudan, "A formal framework for reflective database access control policies," *15th ACM Int. Conf. on Computer and Communications Security*. ACM, Alexandria, Virginia, USA, pp. 289–298, 2008.

[14] L. E. Olson, C. A. Gunter, W. R. Cook, and M. Winslett, "Implementing Reflective Access Control in SQL," *23rd Annual IFIP WG 11.3 Working Conference on Data and Applications Security*. Springer-Verlag, Montreal, P.Q., Canada, pp. 17–32, 2009.

[15] A. Roichman and E. Gudes, "Fine-grained access control to web databases," *12th ACM symposium on Access Control Models and Technologies*. ACM, Sophia Antipolis, France, pp. 31–40, 2007.

[16] Canfora, G.; Visaggio, C.A.; Paradiso, V., "A Test Framework for Assessing Effectiveness of the Data Privacy Policy's Implementation into Relational Databases," in *Intl. Conf. on Availability, Riliability and Security*, 2009, pp. 240–247.

[17] J. Malenfant, M. Jacques, and F. Demers, "A tutorial on behavioral reflection and its implementation," *Proc. Reflect.*, 1996.

[18] B. Hicks, S. Rueda, D. King, T. Moyer, J. Schiffman, Y. Sreenivasan, P. McDaniel, and T. Jaeger, "An architecture for enforcing end-to-end access control over web applications," *15th ACM symposium on Access Control Models and Technologies*. ACM, Pittsburgh, Pennsylvania, USA, pp. 163–172, 2010.

[19] A. Chlipala, "Static checking of dynamically-varying security policies in database-backed applications," in *9th USENIX Conf. on Operating Systems Design and Implementation*, 2010, pp. 1–14.

[20] L. Caires, J. A. Pérez, J. C. Seco, H. T. Vieira, and L. Ferrão, "Type-based access control in data-centric systems," *20th European conference on Programming Languages and Systems: part of the joint European conferences on theory and practice of software*. Springer-Verlag, Saarbrucken, Germany, pp. 136–155, 2011.

[21] S. Chaudhuri, T. Dutta, and S. Sudarshan, "Fine Grained Authorization Through Predicated Grants," *IEEE 23rd ICDE - Int. Conf. on Data Engineering*. Istanbul, Turkey, pp. 1174–1183, 2007.

[22] J. Zarnett, M. Tripunitara, and P. Lam, "Role-based Access Control (RBAC) in Java via Proxy Objects Using Annotations," in *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, 2010, pp. 79–88.

[23] Y. Zhu, H. Hu, G.-J. Ahn, M. Yu, and H. Zhao, "JIF: Java + information flow," 2012. .

[24] H. Garcia-Molina, *Database Systems: The Complete Book*, 2nd E. 2008.

[25] S. Weil, S. Brandt, and E. Miller, "Ceph: A scalable, high-performance distributed file system," *Proc. 7th …*, pp. 307–320, 2006.

[26] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 29–43.

[27] M. Kerzner, "Hadoop Illuminated."

[28] IETF, "RFC 6101: The Secure Sockets Layer (SSL) Protocol Version 3.0."